Analytical Design of Evolutionary Control Flow Components*

Carol L. Hoover
Robotics Institute
Carnegie Mellon University
clh@cs.cmu.edu

Pradeep K. Khosla
Institute for Complex Engineered Systems
Carnegie Mellon University
pkk@ices.cmu.edu

Abstract

The market demands that software systems be adaptable to changes in requirements. Software must be evolvable to solve slightly different problems over time. The transition from real-world requirements to software is a human-intensive and potentially complex process that provides limited automated support for the analysis of alternative designs with respect to their evolvability. In this paper, we propose an analytical software design approach to localize changes to control flow requirements. We present an analytical and "heuristically good" design approach to generate control components that localize change and reduce the computational complexity of an optimal approach. We apply our heuristic to an example and summarize the results. Lastly, we propose future research and summarize our ideas.

1. Introduction

Because the market demands changes in application requirements, there is an increasing need for software systems to adaptively support changes in application-level objectives [3]. A software system should be able to grow and change to solve slightly different problems over time: a software system should be easily changed with minimal error to satisfy changing requirements. Ease and reliability of modification is particularly important for high assurance applications such as air traffic control.

The transition from real-world requirements to software is a human-intensive process with limited support for mathematical and automated analysis of alternative designs. We envision techniques that enable the comparative analysis of alternative designs and trade-offs between design objectives. The goal is to realize a software engineering discipline that increasingly automates the manipulation of analysis and design artifacts.

*Funding for this work comes from U.S. Defense Advanced Research Projects Agency cooperative agreement number F30602-96-2-0240.

First we briefly overview research involving software partitioning and discuss our goals for analytical partitioning that localizes change. Then we introduce an analytical process for determining control flow components that can be changed with minimal effort and error to satisfy plausible changes in control flow. We prove the computational complexity of this process and introduce a heuristic for reducing this complexity. We apply both the optimal approach and the heuristic to an example and compare the results. For the purposes of this paper, we assume an architectural-level view of software components as defined by Garlan and Shaw [5].

2. Software Partitioning and Change

Our overall goal is an analytical way to derive a software architecture that localizes change. The input would be a requirements analysis, known design constraints, and potential changes to the problem or solution. The output would be a specification of software components that should either be selected for reuse or built. We are seeking an analytical way to mathematically and automatically apply Parnas' guidelines for modularizing complex software systems [10].

VanHilst and Notkin applied the language features of C++ to structure class definitions in a way that localizes design decisions and changes to the software solution [11]. The problem is that object-oriented techniques currently guide but do not automate the transformation of requirements into design. Analysis objects do not necessarily map into design objects, and the designer must determine which and how much logic should be encapsulated by each class definition. The resulting decisions may or may not optimize design constraints.

Kazman et al. propose the Software Architecture Analysis Method (SAAM) for analyzing a software architecture with respect to an organization's life cycle concerns, such as efficiency, maintainability, etc. SAAM provides high-level process steps for assessing a particular architecture with respect to design objectives but lets the analyst generate alternative architectures [8].

Assessment of an existing architecture with respect to organizationally significant quality attributes supports our vision of techniques to analyze the fitness of a particular design. We seek techniques that are analytic, automatable, and applicable to the creation of new designs (architectures) as well as to the assessment of existing designs. What we have in mind are the types of analytical techniques that researchers in the area of hardware-software codesign use to partition system functionality [1].

3. Design of Control Flow Components

In this section, we propose the analytical design of software components to direct the flow of control in an application. We assume a software architecture that decouples the complexity of this flow from basic data operations. We present an analytical process to determine the optimal partition of task activations into control components and discuss the strengths and limitations of this process.

3.1. Control Flow Sequences and Change

By control flow, we mean the order of execution of a set of tasks. These could be process control tasks whose order of execution is, for example, <*Read I/O*, *Activate Process*, *Monitor Status*, *Report Status*>. Application requirements may change so that the *Report Status* task should be done immediately after the *Activate Process* task to allow operations that depend upon the activation but not upon the completion of the process to proceed immediately after the process is activated.

We assume the requirements analysis includes a list of potential changes to the required flow. One approach to reduce the complexity of making changes to control flow would be to put task activations in components separate from the tasks. The high-level flow of control would be embedded in control components that activate task components. We use this style of architecture in our study. The idea of decoupling the part of a software solution which may change (in this case the ordering of task activations) from the part not involved in the change (the tasks themselves) is similar to the design pattern in which *Mediator* objects coordinate the application-specific interactions between *Colleague* objects [4].

The problem is how to group the task activations so that the resulting control components can be easily modified to make the potential changes. A process for doing this analytically follows in the next section.

3.2. Analytical Process for Optimal Partitioning

First, we define the terms for our discussion.

- A **sequence S** of tasks, $S = <t_1, t_2,\ldots, t_n>$, refers to the execution order of a set of tasks T, $T = \{t_1, t_2,\ldots, t_n\}$ where *n* is the cardinality of the set.
- The **length of a sequence S**, denoted **| S |**, is the number of task activations in the sequence.
- Given two sequences $X = <x_1, x_2,\ldots, x_n>$ and $Y = <y_1, y_2,\ldots, y_m>$, **X | Y** (read **"X concatenated with Y"**) is a sequence S such that $S = <x_1, x_2,\ldots, x_n, y_1, y_2,\ldots, y_m>$.
- A sequence $X = <x_1, x_2,\ldots, x_m>$ is a **subsequence of S** $= <t_1, t_2,\ldots, t_n>$ if there exists a mapping from X to S such that $x_1 = t_i$, $x_2 = t_{i+1},\ldots, x_m = t_{i+m-1}$ and $X \neq S$. Readers of this paper should note that our definition of a subsequence differs from that in which a sequence $Z = <z_1, z_2,\ldots, z_k>$ is a subsequence of sequence $X = <x_1, x_2,\ldots, x_m>$ if there exists a strictly increasing sequence $<i_1, i_2,\ldots, i_k>$ of indices of X such that for all j = 1, 2,…, k, we have $x_{ij} = z_j$. With this definition, X = <a,c,e> would be a subsequence of Y = <a,b,c,d,e> [2]. With our definition, <a,b,c> is a subsequence of Y; while <a,c,e> is not.
- A **partition P of a sequence S** is a set of subsequences, $\{X_1, X_2,\ldots, X_n\}$ where *n* is the cardinality of the set, such that $S = X_1 \mid X_2 \mid \ldots \mid X_n$. Given that S = <a,b,c>, the set of all partitions is {<a><b><c>, <a><b,c>, <a,b><c>, <a,b,c>}.

In our partitions we would like to preserve the required order of task activations in the control components. It would not be meaningful for us to group a required order of task activations <a,b,c> into control components containing <a,c> and <b>. So by our definition, <a,c><b> is not a partition of <a,b,c>; and <a,c> is not a subsequence of <a,b,c>.

The following step-wise process analytically generates the control flow components that will optimally reduce the complexity of evolving the required control flow.

1. Develop a metric for determining the difficulty and "error-proneness" of modifying the control components to make a required change in control flow.
2. Determine the way in which the control components would have to be modified and relate this process to the metric. In other words, determine how the

modification steps relate to the counting that must be done for the metric.

3. Express the required control flow as a sequence and generate all partitions of this sequence.
4. For each partition and for each potential change to the required control flow, "walk-through" the necessary modifications to make the change and apply the metric. Save these values in a table that stores the change complexity value for each partition with respect to every potential change.
5. For each partition, sum the change complexity values to derive the total across all potential changes. The partition with the minimum total is the best design choice for the required control flow and potential changes.

To demonstrate this process, we present an example in the next section.

3.3. Application of the Analytical Process

Our previous analysis of genetic algorithms demonstrated the use of master control logic to activate operations performed by reusable components such as the search space, the fitness values, a random item selector, and a value-based item selector [7]. The required sequence of application-level tasks might be <A,B,C,D,E,F,G,H,I>, where each letter shown below represents an activation of a specific task for a genetic algorithm.

A Initialize the population.
B Retrieve population member 1.
C Retrieve the fitness value for member 1.
D Randomly select population member 2.
E Randomly select a genetic operator.
F Apply the genetic operator to generate a new member.
G Determine the fitness value for the new member.
H Select the most fit of member 1 or the new member.
I Save the most fit of member 1 or the new member.

For this example, we propose the alternative sequences <A,D,E,B,C,F,G,H,I> and <A,E,B,C,D,F,G,H,I>. Now we apply our analytical process to design the control logic.

Step 1: Develop the change complexity metric.

Our metric is based upon the following rationale.

- The human does modifications to software.
- Localized change involves fewer components and less effort than non-localized change.
- Small components are easier to understand and modify correctly than are large, complex components.

A basic way to determine the effect of a change is to count the number of task activations contained in the modified control flow components.

Figure 1: Change complexity metric

Given x is a component to be modified, let $|x|$ be the number of task activations in x or the size of x. Let X be the set of all x.

$$\text{Change complexity metric} = \sum_{x_i \in X} |x_i|.$$

Step 2: Determine the process for modifying control components and relate it to the change complexity metric.

We use a master component to activate the other control components in the proper order. Alternatively, the control components could trigger each other. For each control component that must be changed to implement the new control sequence, we sum the number of task activations in the component.

Step 3: Express the required control flow as a sequence and generate all partitions of this sequence.

The required sequence is <A,B,C,D,E,F,G,H,I>. Partitions of the sequence are:

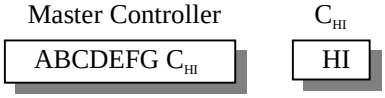
1) <A><B><C><D><E><F><G><H><I>
2) <A><B><C><D><E><F><G><H,I>
3) <A><B><C><D><E><F><G,H><I>
4) <A><B><C><D><E><F><G,H,I> …
256) <A,B,C,D,E,F,G,H,I>.

Steps 4 & 5: Determine the necessary modifications to make each potential change, apply the metric, and store the change complexity values in a table. For each partition, sum the values to derive the total across all potential changes.

Partition <A><B><C><D><E><F><G><H><I> is implemented as a master control component, MC, containing all of the task activations. Changing the control sequence to <A,D,E,B,C,F,G,H,I> requires changes to the master controller. The size of the master controller is 9 because it contains 9 task activations, so the change complexity is 9.

| Master Controller | = 9

The partition <A><B><C><D><E><F><G><H,I> is implemented as shown below.



| Master Controller | = 8 | C_{HI} | = 2

To strictly follow our two-level control flow architecture, we would place the singleton task activations each in a separate component. But placing these task activations directly in the master controller does not affect the overall change complexity (because the size of MC would be the same) and eliminates the redundancy of "activating a component which activates precisely one task ".

The change complexity is determined for each partition and for each change as shown in the table below. There are two alternative sequences considered for this example. Columns 2 and 3 of Table 1 show the complexity of changing from the required sequence <A,B,C,D,E,F,G,H,I> to an alternative sequence <A,D,E,B,C,F,G,H,I> and from <A,B,C,D,E,F,G,H,I> to <A,E,B,C,D,F,G,H,I>.

Table 1: Complexity Per Partition and Alternative	Alter. 1	Alter. 2	Total
<A><B><C><D><E><F><G><H><I>	9	9	18
<A><B><C><D><E><F><G><H,I>	8	8	16
<A><B><C><D><E><F><G,H><I>	8	8	16
…	…	…	…
<A><B,C><D><E><F,G,H,I>	5	5	10
…	…	…	…
<A><B,C><D,E><F,G,H,I>	4	6	10
…	…	…	…
<A,B,C,D,E,F,G,H,I>	9	9	18

We leave it to the reader to complete the table for all 256 feasible partitions and to determine the partition(s) with the lowest complexity value.

3.4. Complexity of the Process

The number of partitions of a sequence S increases exponentially with the length of the sequence: the cardinality of the set of all partitions is 2^{n-1} where *n* is the length of S. For small values of *n*, this may not be a problem. But as *n* increases, the execution time required to find the optimal partition would become computationally expensive *($O(2^n)$)*.

One could apply a combinatorial optimization algorithm such as a genetic or simulated annealing algorithm [6] [9]. The change complexity metric would provide the rationale for an objective function. The primary issue would be how to automate the process of "walking-through" the change to a control flow sequence in order to determine the change complexity value.

This appears to be an intricate process. If we could constrain our design objectives so that we would not be required to generate all possible partitions, we might achieve a polynomial-time algorithm for determining an appropriate partition of the required control flow into components. The subject of the next section is a heuristic for constraining the design space.

Figure 2. Proof of number of partitions

We will prove inductively our hypothesis that the number of partitions for a sequence of length *n* is 2^{n-1}. Let $X_1 = \langle x_1 \rangle$ and $X_2 = \langle x_1,x_2 \rangle$ be the base sequences. The length of X_1 is 1, and the length of X_2 is 2. X_1 can be partitioned in exactly one way, $\langle x_1 \rangle$. The number of partitions of X_1 is $1 = 2^{(1-1)} = 2^{n-1}$ for $n = 1$. X_2 can be partitioned in two ways, $\langle x_1 \rangle\langle x_2 \rangle$ and $\langle x_1,x_2 \rangle$. Therefore the number of partitions of X_2 is $2 = 2^{(2-1)} = 2^{n-1}$ for $n = 2$. We assume that the number of partitions for a sequence X_j of length *j* is 2^{j-1}.

We will now show that our inductive hypothesis is true for sequences of length *j+1*. Let us form a sequence X_{j+1} by concatenating the sequence X_1 with sequence X_j, $X_1 \mid X_j$. The length of X_{j+1} is the length of X_1 plus the length of X_j or *j+1*. There are strictly two ways to form partitions of X_{j+1}. One way is to group x_1 by itself. The number of possible ways to group the other elements of X_{j+1}, namely those contributed by X_j, would be the number of partitions of X_j or 2^{j-1}. The other way is to group x_1 with the first element of X_j. Likewise, the number of partitions containing x_1 that are grouped with the first element of X_j is the number of partitions of X_j or 2^{j-1}. So the total number of partitions of X_{j+1} is $2^{j-1} + 2^{j-1} = 2^j = 2^{n-1}$ for $n = j+1$. Hence, the inductive hypothesis holds for sequences of length *j+1*.

4. "Heuristically Good" Partitioning Process

The method described in the previous section enumerates all valid partitions for a given sequence and identifies those that optimize the complexity of making the required changes in control flow. In this section, we describe a partitioning pattern that tends to minimize the overall change complexity. We prove the "*goodness*" of this pattern for most potential changes to control flow and explain the exception. Then we present a process for determining control components that exhibit this pattern.

4.1. Pattern for Process Improvement

An invariant subsequence, such as <B,C> or <F,G,H,I> in our genetic algorithm example, that is located in a single component helps to reduce the overall

change complexity. We can more formally define an *invariant subsequence* and the *longest invariant subsequence* as follows.

Let:

- S = $\{s_i\}$ = the set of plausible sequences of task activations, where $s_i = <t_{i1},\ldots, t_{in}>$ and $t_{ij} \neq t_{ik}$ for every $j \neq k$.
- T = $\{t_i\}$ is the set of application-level tasks. If $v = <t_1,\ldots, t_m>$ with $1 < m \leq n$, then Invariant(v) $\leftrightarrow \forall s_i, \exists j: t_{ij} = t_1,\ldots, t_{i(j+m-1)} = t_m$.
- LI, the set of longest invariant subsequences, = $\{l_j\}$: Invariant(l_j) $\wedge \neg (\exists l_k$: Invariant(l_k) $\wedge$ l_j = Subsequence(l_k)).

Three questions need to be answered.

1. Why does the placement of an invariant subsequence in a single component make it easier to make anticipated changes to the control flow?
2. How does one locate an invariant sequence automatically?
3. How can the location of invariant sequences be combined with the optimal partitioning approach?

The answer to the first question is the topic for the next section.

4.2. Proof of Heuristic

The idea is to compare the type of partition in which an invariant subsequence Y is placed in a single component to other possible types of partitions in which Y is not located in a single component. We will show that placing Y in a single component results in a lower or equally good complexity value for all types of partitioning except one.

Let:

- S = <X, Y, Z> be a sequence of task activations with subsequences X, Y, and Z.
- Y be a longest invariant subsequence in S. Y = $<y_1,\ldots, y_{|Y|}>$, $|Y| \geq 2$.
- P(X) be a partition of a subsequence X.
- X = <X', X"> and Z = <Z', Z">, where any of X', X", Z', or Z", may be empty subsequences.
- MC_j represent the *case j* master controller.
- [X] represent a component containing a subsequence X, and [P(X)] represent the components containing a partition of X.
- C([X]), the change complexity for a component containing a subsequence X, be: 0, if the component need not be changed, or |[X]| = |X|, the size of the component containing X which equals the length of X, if the component must be changed. C([MC_j]) = |MC_j|.
- C(P[X]) = $\sum$ C([x_i]) where $x_i \in$ P[X].

There are six basic ways to partition Y.

Case 1: [P(X)] [Y] [P(Z)]
Y is in a single component by itself.

Case 2: [P(X')] [X"Y] [P(Z)] or [P(X)] [YZ'] [P(Z")]
Y is located solely within either [X"Y] or [YZ'].

Case 3: [P(X')] [X"YZ'] [P(Z")]
Y is located solely within [X"YZ'].

Case 4: [P(X)] [P(Y)] [P(Z)]
Y is partitioned across two or more components but is not contained within [P(X)] or [P(Z)].

Case 5: [P(X')] [X"$y_1\ldots y_j$] [P($y_{j+1}\ldots y_n$)] [P(Z)] or
[P(X')] [X"$y_1\ldots y_j$] [P($y_{j+1}\ldots y_k$)] [$y_{k+1}\ldots y_n$Z'] [P(Z")] or
[P(X)] [P($y_1\ldots y_j$)] [$y_{j+1}\ldots y_n$Z'] [P(Z")]
Y is located within but not solely within [X"$y_1\ldots y_j$], [$y_{k+1}\ldots y_n$Z'], or [$y_{j+1}\ldots y_n$Z'].

Case 6: [P(X')] [X"$y_1\ldots y_j$] [$y_{j+1}\ldots y_n$Z'] [P(Z")]
Y is partitioned across X and Z and is solely within [X"$y_1\ldots y_j$] and [$y_{j+1}\ldots y_n$Z'].

Cases 1-3 are variants of our targeted partitioning pattern in which Y is contained in a single component. Now we will show that at least one of the *Cases 1-3* type partitions performs better or equally as well as *Cases 4-5* type partitions and, in certain situations, as well as *Case 6* type partitions. From here on, *Case X* refers to *Case X* type partitions unless specified otherwise. The reader should note that because Y is invariant, the C([Y]) and C([P(Y)]) are always zero.

Comparison with Case 4:

Some changes to S must involve [P(X)] or [P(Z)] since Y is invariant. If these changes do not involve the master controller, then the relevant change complexities are the same for *Case 1* and *Case 4*: C([P(X)]) + C([P(Z)]).

Now suppose that changes to [P(X)] and [P(Z)] do require changes to the master controller, then the resultant change complexity expressions are:

Case 4: C([P(X)]) + C([P(Z)]) + C([MC_4]).

Case 1: C([P(X)]) + C([P(Z)]) + C([MC_1]).

C([MC_1]) < C([MC_4]) because Y is contained in one component for *Case 1* but in two or more components for *Case 4*. In this situation, locating Y in a single component is better than partitioning it across several components that contain subsequences of Y.

Comparison with Case 5:

As in *Case 4*, some changes to S must involve [X"y_1…y_j] and [y_{k+1}…y_nZ'] since Y is a longest invariant subsequence. The related change complexities for *Case 5* and *Case 1* are as follows.

Case 5: C([P(X')]) + C([X"y_1…y_j]) + C([P(Z)]) + C([MC_5]).

Case 1: C([P(X')]) + C([X"]) + C([P(Z)]) + C([MC_1])

or

Case 5: C([P(X')]) + C([X"y_1…y_j]) + C([y_{k+1}…y_nZ']) + C([P(Z")]) + C([MC_5]).

Case 1: C([P(X')]) + C([X"]) + C([Z']) + C([P(Z")]) + C([MC_1]).

or

Case 5: C([P(X)]) + C([y_{j+1}…y_nZ']) + C([P(Z")]) + C([MC_5]).

Case 1: C([P(X)]) + C([Z']) + C([P(Z")]) + C([MC_1]).

Suppose these changes do not involve the master controller. *Case 5* changes that involve components containing Y subsequences are more expensive than the changes to the corresponding *Case 1* partition because the affected *Case 1* components do not contain elements of Y. All other *Case 5* changes are equally as expensive as the corresponding *Case 1* changes. Hence, the overall change complexity for *Case 1* is less than that for *Case 5* when the master controller is not involved.

Suppose changes to S do involve the master controller. C([MC_1]) <= C([MC_5]) because MC_1 contains one activation for the component containing Y; whereas MC_5 contains one or more activations for components encapsulating only subsequences of Y. Therefore with *Case 5*, changes involving the master controller not only incur extra costs due to Y subsequences contained in the affected components but also due to a potentially larger master controller. *Case 1* performs as well as or better than *Case 5*.

Comparison with Case 6:

Again, some changes to S must involve [X"y_1…y_j] and [y_{j+1}…y_nZ'] since Y is a longest invariant subsequence. Suppose these changes do not involve the master controller. The resultant change complexities are:

Case 6: C([P(X')]) + C([X"y_1…y_j]) + C([y_{j+1}… y_nZ']) + C([P(Z")]).

Case 1: C([P(X')]) + C([X"]) + C([Z']) + C([P(Z")].

As in *Case 5*, *Case 1* has a lower complexity value because no component that must be changed contains subsequences of Y.

Now suppose that changes to S involve the master controller. The resultant change complexities are:

Case 6: C([P(X')]) + C([X"y_1…y_j]) + C([y_{j+1}… y_nZ']) + C([P(Z")]) + C([MC_6]).

Case 1: C([P(X')]) + C([X"]) + C([Z']) + C([P(Z")] + C([MC_1]).

In this situation, we observe a complication. C([MC_1]) = C([MC_6]) + 1 because MC_1 contains an extra activation for [Y]. For changes that affect [X"y_1…y_j] or [y_{j+1}…y_nZ'], *Case 1* is still equally or more efficient than *Case 6*. The size of these components must be at least one greater than their *Case 1* counterparts [X"] and [Z']: a fact that outweighs the slightly larger MC_1. But for the changes that do not affect [X"y_1…y_j] or [y_{j+1}…y_nZ'], *Case 1* performs less efficiently because of the larger MC_1. The effects of this minor size difference may be significant depending upon the number of potential new sequences which involve changes to the master controller but no changes to [X"y_1…y_j] or [y_{j+1}…y_nZ'].

One might think that *Case 2* or *Case 3* could always perform as efficiently as *Case 6* because they do not increase the size of the master control component. In fact, both of these cases reduce the size of the master controller by at least 1. For changes that do not affect [X"Y], [YZ'], or [X"YZ'], *Case 2* and *Case 3* perform as well as or better than *Case 6*. But for changes that do affect these components, they may perform less efficiently because the Y subsequence is solely contained in a single component. If that component is frequently involved in the required changes, then *Case 2* or *Case 3* may be less efficient than *Case 6*.

In *Case 6*, we observe a trade-off between the size of the master control component, the size of the control flow components involved in changes to control flow, and the actual changes needed to produce the potential new control sequences. A brute force method of enumerating all *Case 1-3* and *Case 6* partitions and summing the corresponding total complexities across all potential new control flows appears to be the only way to obtain an optimal partition for these special situations. On the other hand, *Case 1* is always better than *Cases 4* and *5* and can even outperform *Case 6* when the master controller is not involved or when the components containing Y subsequences are included in the change. *Case 1* type partitions are "heuristically good" but not optimal.

4.3. Process for Applying the Heuristic

To efficiently apply our heuristic, we need a polynomial-time algorithm for locating the longest invariant subsequences across all expected or potential permutations of the original control flow sequence. We also need a process for partitioning the remaining variant subsequences.

In this section, we discuss an *O(m*n)* algorithm for locating the longest invariant subsequences, where *m* is the number of alternative sequences and *n* is the length of the required sequence. A bitmap represents *the immediately preceding relationship* between the task activations across the alternative control sequences. A one in the j-th bit position means that the j-th task activation in the required control flow sequence immediately precedes the (j+1)-th task activation in the required sequence across all projected permutations. The reader should note that the n-th task has no successor; hence, the bitmap contains a zero in the n-th bit.

The logic of the algorithm is to scan each alternative control flow sequence (the list of potential sequences from the requirements analysis) and to locate each task activation which is **not** followed by the same activation that succeeds it in the required sequence. When such a case is found, the bit for that task activation is set to zero in the bitmap. After all alternative sequences are scanned, the bitmap contains the information needed to locate the longest invariant sequences.

The algorithm then scans the bitmap by starting from the first bit (which corresponds to the first task activation in the required sequence) and moving across the bitmap to the last bit (which corresponds to the last task activation in the required sequence). As it scans, the algorithm records task activation subsequences associated with strings of 1's and 0's in the bitmap. Strings of 1's (plus the next immediate 0-bit) represent longest invariant subsequences. Strings of 0's (not including the 0 ending a string of 1's) represent longest variant subsequences.

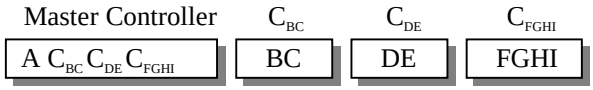
The next step is to place each longest invariant subsequence in a separate component. The optimal method can be used to partition the remaining task activations. For simplicity, we place each non-singleton variant subsequence in its own component. Singleton task activations are placed in the master controller.

Below we apply the algorithm to our previously discussed example.

Algorithmic Step **Complexity**

1. Initialize bitmap to [1|1|1|1|1|1|1|1|0] for the required sequence <A,B,C,D,E,F,G,H,I>. *O(n)*
2. Scan the *m* alternative control sequences to find task activations that do not precede the same task activation that they precede in the required sequence. *O(m* n)*
   For each of m sequences
   For each of n task activations
   Does the task precede a different task from that which it precedes in the required task?
   If yes, set its position in the bitmap to zero.
   Endfor
   Endfor
   The bitmap is altered as follows while scanning the alternative sequence <A,D,E,B,C,F,G,H,I>.
   Evaluate A. Set A-bit to 0 in [0|1|1|1|1|1|1|1|0].
   Evaluate D. Do not alter bitmap.
   Evaluate E. Set E-bit to 0 in [0|1|1|1|0|1|1|1|0].
   Evaluate B. Do not alter bitmap.
   Evaluate C. Set C-bit to 0 in [0|1|0|1|0|1|1|1|0].
   Evaluate F. Do not alter bitmap.
   Evaluate G. Do not alter bitmap.
   Evaluate H. Do not alter bitmap.
   Evaluate I. Do not alter bitmap.
   The bitmap after analyzing <A,D,E,B,C,F,G,H,I> is [0|1|0|1|0|1|1|1|0]. Analyze <A,E,B,C,D,F,G,H,I> starting with the current contents of the bitmap. The bitmap after analyzing <A,E,B,C,D,F,G,H,I> is [0|1|0|0|0|1|1|1|0].
3. Scan the bitmap to locate the invariant and variant subsequences. *O(n)*

The ones-strings or invariants are <B,C> and <F,G,H,I>. The zero-strings are <A> and <D,E>. The resulting "heuristically good" partition would be <A><B,C><D,E><F,G,H,I>. The corresponding control components are shown below.



5. Future Research

We are currently investigating subsequences that vary internally but not externally. Preliminary results show that a good heuristic may be to place this type of subsequence within a single component. We plan to analytically compare this heuristic with respect to the partitioning types presented in this paper.

One variation to our change complexity metric would be to determine the complexity for a sequence of new control flows. Given a sequence $X = <x_1, x_2,\ldots, x_m>$ in which each x_i, $1 \le i \le m$, is a permutation of a set of tasks $T = \{t_1, t_2,\ldots, t_n\}$, where *n* is the cardinality of the set, one could determine the change complexity metric for X. This would be the total complexity of changing x_1 to x_2, x_2 to x_3,…, and x_{m-1} to x_m. Here, x_1 represents the required control flow; while x_2, x_3,…, and x_m represent the alternative control flows. In this paper, each change started from the required control flow.

Another variation is to assign a probability value to each potential control flow being considered during design. The change complexity metric might appear as follows.

Figure 3. Alternate change complexity metric

$$\text{change complexity metric} = \sum_{x_i \varepsilon X} w_i x_i$$

where X is the set of change complexity values for a particular partition with respect to each potential change, $w_i > 0$ is the likelihood that the potential change associated with the change complexity value x_i will be made, and $\sum w_i = 1.0$.

The validity of any metric depends upon how well it measures the targeted behavior. In our work, the targeted behavior was modification of control flow components within a master control architecture. A study of the type of cognitive tasks that programmers perform in maintaining and evolving software systems may also help us to develop more comprehensive metrics [12].

6. Summary

The goal of this paper is to motivate the need for analytical techniques to support the design of software that can be changed easily with minimal error. To demonstrate this need, we discussed the complexity of changing software components that localize changes to control flow. We presented an optimal partitioning process for deriving components that minimize the complexity of making anticipated changes to control flow.

Because of the exponential complexity of this "*brute force*" approach, we proposed a "*heuristically good*" partitioning technique in which the longest invariant subsequences of the original control flow sequence are each located in a single component. We listed six basic ways to partition the required control flow sequence, three of which located a given longest invariant subsequence Y in a single component.

Through comparative analysis, we demonstrated that our heuristic improves or matches the complexity of change with respect to all other partitioning types except one. In the exceptional situation, *Case 6* in *Section 4.2*, Y is spread across two components, each of which embeds other task activations with part of Y.

The heuristic outperforms *Case 6* when no changes to the master controller are required. Otherwise, the trade-off between the size of the master controller, the sizes of the control components that must be changed, and the types of required changes across all new potential control flows may favor *Case 6*. Research involving subsequences that are internally variant but externally invariant may yield a polynomial-time selection of a suitable partition for *Case 6* exceptions.

Lastly, we presented an algorithm for locating longest invariant subsequences in a required control flow sequence. We applied it to a genetic algorithm example and summarized our results. In summary, our goal is to motivate the reader to consider an analytical technique for designing control flow components.

7. References

[1] J.K. Adams and D.E. Thomas, "Design Automation for Mixed Hardware-Software Systems", *Electronic Design*, Vol. 45, No. 5, Mar. 1997, pp. 64-66, 71-72.

[2] T.H. Cormen, C.E. Leiserson, and R.L Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1990.

[3] M. Fayad and M.P. Cline, "Aspects of Software Adaptability", *Communications of the ACM*, Vol. 39, No. 10, Oct. 1996, pp. 58-59.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Pub. Co., Reading, Mass., 1994.

[5] D. Garlan and M. Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, World Scientific Pub. Co., 1994.

[6] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Pub. Co., Reading, Mass., 1989.

[7] C.L. Hoover and P.K. Khosla, "An Analytical Approach to Change for the Design of Reusable Real-Time Software", *Proc. of the Second Workshop on Object-Oriented Real-Time Dependable Systems* (WORDS'96), IEEE Computer Society Press, Los Alamitos, Calif., Feb. 1-2, 1996, pp. 144-151.

[8] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-Based Analysis of Software Architecture", *IEEE Software*, Vol. 13, No. 6, Nov. 1996, pp. 47-55.

[9] S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi, "Optimization by Simulated Annealing", *Science*, Vol. 220, No. 4598, May 13, 1983, pp. 45-54.

[10] D.L. Parnas, P.C. Clements, and D.M. Weiss, "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, Mar. 1985, pp. 259-266.

[11] M. VanHilst and D. Notkin, "Decoupling Change from Design", *Software Engineering Notes*, Vol. 21, No. 6, Nov. 1996, pp. 58-69.

[12] A. von Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution", *Computer*, Vol. 28, No. 8, Aug. 1995, pp. 44-55.